Green Analysis and Visibility User's Guide (GreenAV v.3.0.0)

Developed by
Christian Schröder
and Wolfgang Klingauf and Robert Günzel
Technical University of Braunschweig, Dept. E.I.S.

8th July 2009

Contents

1 Introduction **4**

2 Green Analysis and Visibility **5**

2.1 Architecture 5

2.2 Namespace gs::av 6

2.3 API adapter GAV_Api 7

2.4 Output Plugins 7

2.4.1 Default Output Plugin (special case) 15

2.4.2 NULL Output Plugin (special case) 15

2.4.3 STDOUT Output Plugin 15

2.4.4 Text-file Output Plugin 15

2.4.5 Comma Separated Values (CSV) Output Plugin 16

2.4.6 SCV-stream Output Plugin 17

2.4.7 How to implement an Output Plugin 19

2.5 Analysis and Visibility Service 21

2.5.1 Service Plugin 21

2.5.2 Commands 21

2.5.3 Event Listener 22

2.6 Statistics Calculator 23

2.6.1 Trigger 26

2.6.2 Calculator 31

2.6.3 StatCalc Miscellaneous 37

2.6.4 How to use Overview 38

2.7 Miscellaneous 38

2.7.1 Dynamic Processes 38

2.7.2 SystemC 2.1 39
2.7.3 Regression Tests 39

A Appendix **40**

A.1 Green Analysis and Visibility Requirements 40
A.1.1 General 40
A.1.2 Details 40
A.1.3 Nice to have features 41

List of Figures

2.1 GreenAV Concept 5
2.2 GreenAV full Concept 6
2.3 GreenAV Concept for Outputs: The `GAV_Plugin` contains several types of output plugins 8
2.4 GreenAV Output Example 14
2.5 CSV ouput plugin Excel example 18
2.6 GreenAV Statistics Calculator Concept 25

Chapter 1

Introduction

The *Green Analysis and Visibility* (GreenAV) framework is an extension service for the GreenControl framework.

GreenAV makes wide use of the GreenControl's configuration service GreenConfig. Please see the GreenControl User's Guide for an introduction to GreenControl and the GreenConfig User's Guide for details about GreenConfig.

GreenAV extends configurable parameters with analysis and visibility features. It is possible to combine several parameters within a formula doing mathematical calculations and statistics with flexible triggers for the calculations. It is also possible to output parameters and analysis results to several output plugins which make them available at runtime to make them visible or store them e.g. in files of different kinds. These outputs can be connected to ESL vendor tools (e.g. via SCV-streams).

Visit the GreenSocs web page to get the newest release of the GreenControl framework: http://www.greensocs.com/projects/GreenControl.

Visit the GreenAV project web page to get the newest development information and further documentation: http://www.greensocs.com/projects/GreenControl/GreenAV.

Chapter 2

Green Analysis and Visibility

2.1 Architecture

Green Analysis and Visibility (GreenAV, GAV) follows the GreenControl concept (see figure 2.1): A plugin (*GAV_Plugin*) manages the functionality that is needed centralized for analysis and visibility. An API (*GAV_Api*) allows access to the plugin's functionality.

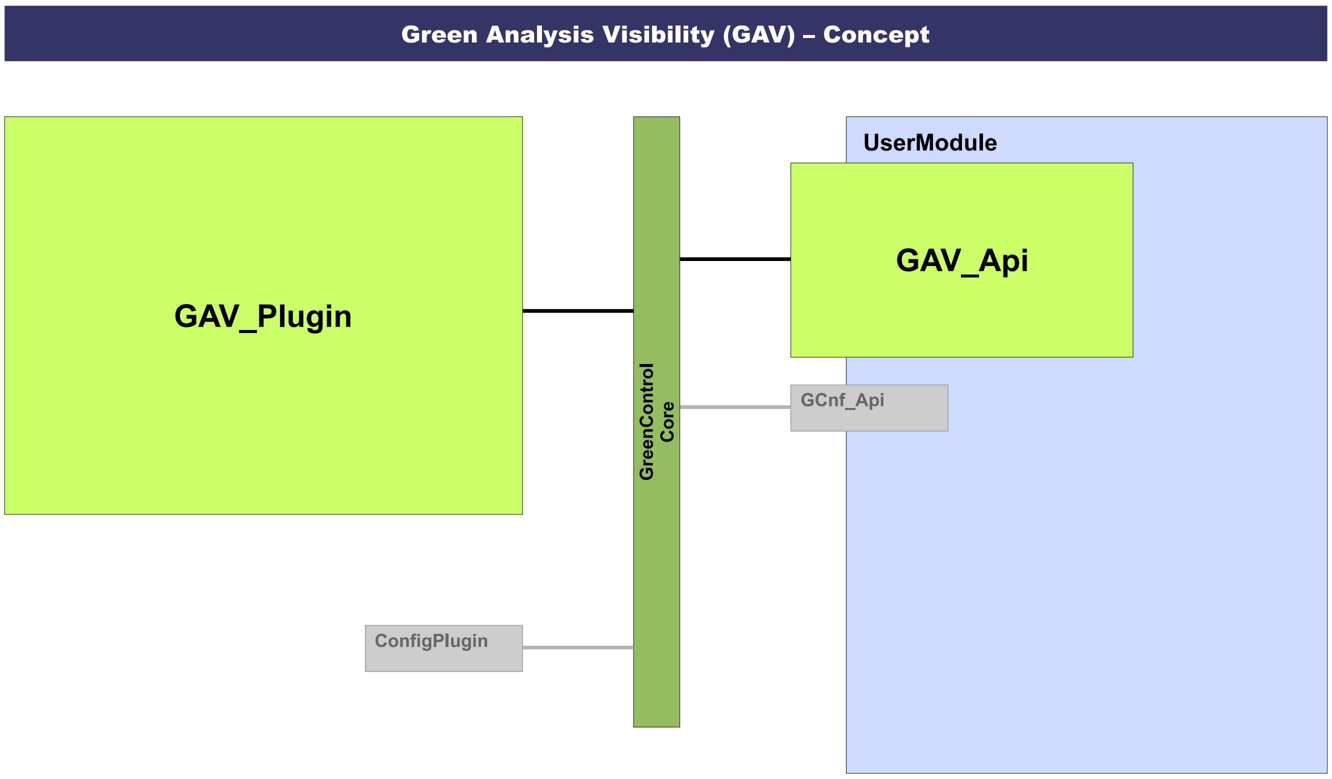


Figure 2.1: GreenAV Concept

Figure 2.2 is an overview over the most important elements of GreenAV. The user module may use the `GAV_Api` to access `OutputPlugins` (see section 2.4). The user module also can create Statistics Calculator objects `StatCalc` and the concerning `Triggers` and `Calculators` (see section 2.6). The event listeners are members of the `GAV_Plugin`. One is used by the output plugins and one is used by the triggers without the user's recognition.

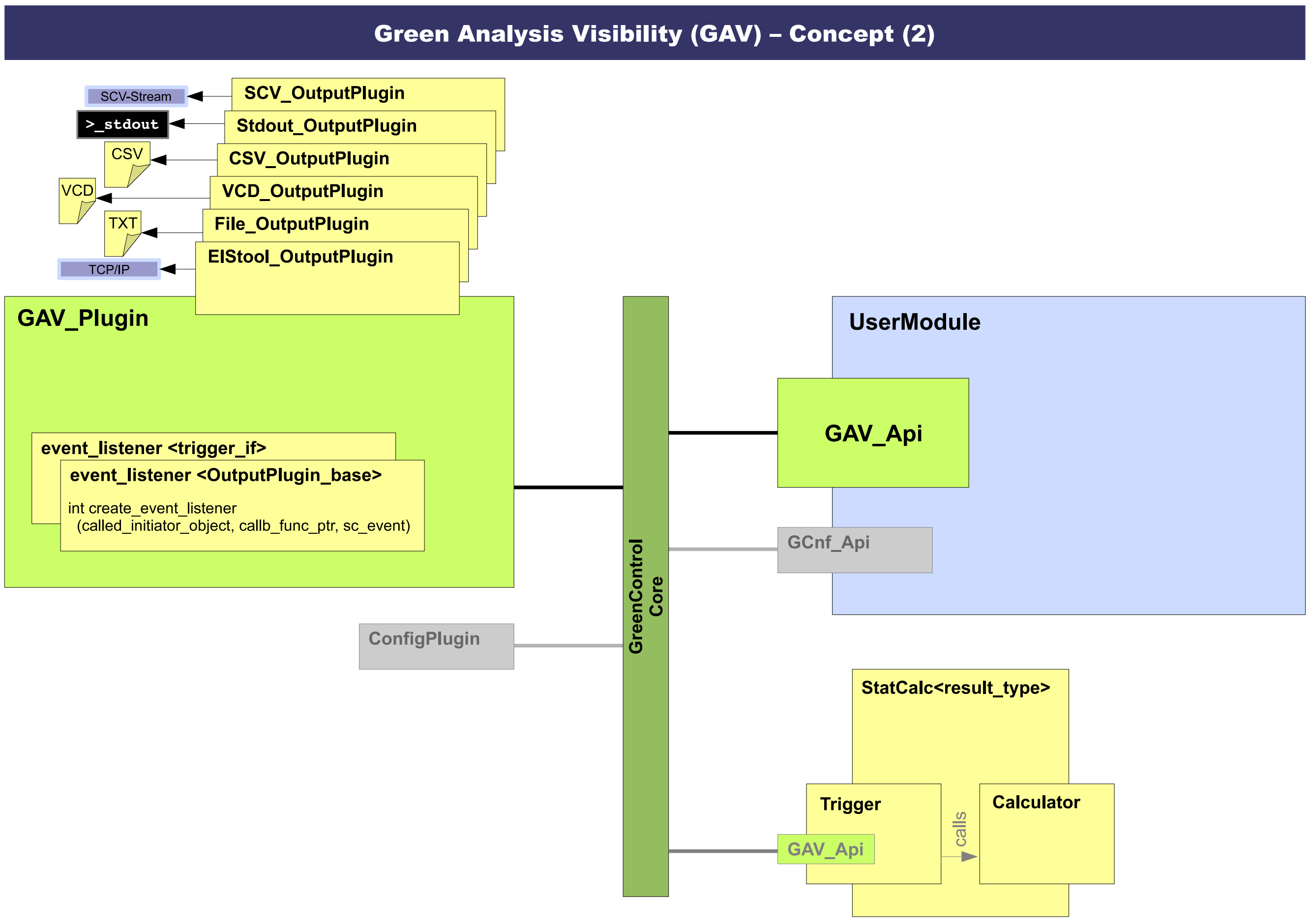


Figure 2.2: GreenAV full Concept

2.2 Namespace gs::av

All analysis and visibility classes of this GREENAV framework are located within the namespace gs::av which is a sub namespace of the GreenSocs namespace gs.

This framework uses the GREENCONTROL namespace gs::ctr and some elements of the GREENCONFIG namespace gs::cnf.

A `using namespace ctr;` statement in the GREENAV global import file imports the control namespace to the gs::av namespace (see greencontrol/gav/plugin/gav_globals.h).

Compatibility Note

Namespace compativility to release 0.2

To be compatible to the old namespaces (tlm::gc, tlm::gc::config) the header file greencontrol/namespace_compatibility.h can be included!

2.3 API adapter GAV_Api

This section is about the GAV user API. For a full API reference see the doxygen API reference. Here follows a brief description of the most important functions:

create_OutputPlugin (OutputPluginType type, string ctor_param)
Creates a new OutputPlugin instance of the specified type in the plugin and provides the specified constructor parameter. The constructor parameter is interpreted dependent on the output plugin type (e.g. a file name or stream name or ignored).
Returns a pointer to the output plugin.
See section 2.4 for details.

get_default_output (OutputPluginType type = DEFAULT_OUT)
Get the default output plugin of the specified type or the simulation-wide default. Not specifying the OutputPluginType will return the simulation-wide default one.
Returns a pointer to the default output plugin.
See section 2.4 for details.

add_to_default_output (OutputPluginType type, gs_param_base *par) add_to_default_output (OutputPluginType type, gs_param_base &par)
Adds a gs_param to be outputted by the (implicit existing) default output plugin of the specified type. The type DEFAULT_OUT may be used to add to the simulation-wide output plugin. For each OutputPluginType type there is existing one object in the GAV plugin. (The object will be created at its first usage.)
Returns a pointer to the output plugin.
See section 2.4 for details.

add_to_output (OutputPlugin_if* outputPluginID, gs_param_base *par)
Adds a gs_param to be outputted by the specified output plugin. The specified output plugin has to be created by calling `create_OutputPlugin` or has to be set by calling `get_default_output`.
See section 2.4 for details.

get_event_listener ()
Returns an event listerner for triggers. May be used for any further purposes.
See sections 2.5.3 and 2.6.1 for details.

2.4 Output Plugins

Output plugins are used to export any `gs_param` value over time to a specific output. Such outputs can be the stdout, text-files, csv-files, SCV streams, dedicated connections to tools, e.g. TCP/IP connections etc. (see Figure 2.3).

Each output plugin is able to output multiple `gs_params` which can be added to the output during simulation runtime by a user or a tool. The only assumption is to have access to a GAV User API (see section 2.3).

Before you are able to use one or more output plugin(s), you have to include the one(s) you need:

```
// e.g.
#include "greencontrol/analysis.h" // includes the Standard Output Plugin
#include "greencontrol/analysis_file_outputplugin.h"
#include "greencontrol/analysis_csv_outputplugin.h"
#include "greencontrol/analysis_scv_outputplugin.h"
// etc.
```

Listing 2.1: Include for output plugin

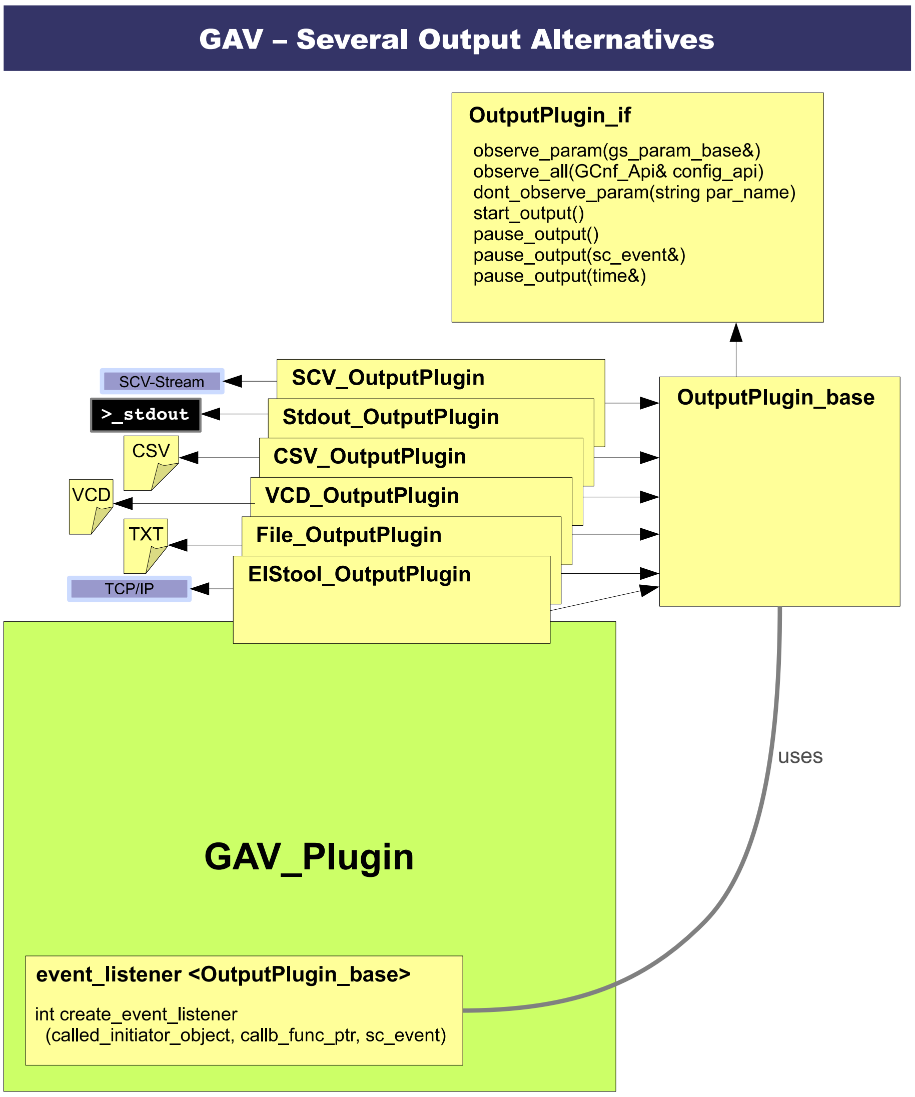


Figure 2.3: GreenAV Concept for Outputs: The `GAV_Plugin` contains several types of output plugins

Default Output Plugin(s)

The provided output plugins are described in the following subsections. Any output plugin existing in the simulation is owned by the GAV plugin. For each type of output plugin there is one *default instance* which should be used if (only) one instance (instead of multiple ones) meets the user's requirements. A simulation-wide default output plugin type (which results in a default instance) should be set by the user during construction. If more than the default output(s) is needed (e.g. if several files should be created) the user may create *additional instances* of each output plugin type.

Specify the simulation-wide default output plugin like shown in listing 2.2. All outputs that are not directed to a special type or instance will use this default output plugin.

The output plugin types are identified with the type gs::gav::OutputPluginType (which is an unsigned int). For convenience their names can be used instead. See the following sections of each output plugin for its value. DEFAULT_OUT is used as a replacement for the simulation-wide default.

```
gs::av::GAV_Plugin analysisPlugin("AnalysisPlugin", gs::av::STDOUT_OUT);
```

Listing 2.2: Specify the default output plugin during Analysis Plugin instantiation

⚠ Only if you know what you are doing, you may use the GAV Plugin function `void set_default_output_plugin(OutputPluginType default_output_plugin_type)` to modify the default output plugin even during model creation and simulation.

See next paragraph and section 2.3 for how to access default output plugins and how to create and access additional ones.

Implementation Note

Output Plugin identification

The output plugins are identified (id) using the unsigned int OutputPluginType. A static map OutPluginName::getMap() is used to store the names (values) related to the id (key). A variable of type OutputPluginType, named as the output plugin name, with the corresponding value can be used to identify the type using gs::av::TXT_FILE_OUT etc.

Access Output Plugins

Before using an output plugin the user needs to get a default instance or create a new one. Use the GAV user API (see section2.3). Several functions are available to access output plugins. See the code example for an example how to use.

- Recommended way accessing the default output plugin of the needed type:
  Get the default output plugin by calling `OutputPlugin_if* get_default_output(const OutputPluginType type = DEFAULT_OUT)`. If the module does not care about the output type, leave the type empty which will return the simulation-wide default output plugin. The returned output plugin pointer can be used to add parameters for observation or to control it (see section Interface and how to use).

- When needing further output plugin instances, a call of `OutputPlugin_if* create_OutputPlugin (OutputPluginType, string ctor_par)` will create a new instance using the string as constructor parameter (e.g. filename). The user should store the returned pointer to be able to add parameters using the function `add_to_output` (see below).

Observing Parameters with Output Plugins

There are two basically different ways adding a parameter for observation to an output plugin: One way is calling GAV user API functions. This assumes you have pointer access to the output plugin instance and the parameter (which is no issue within the simulation). Another way is creating and manipulating parameters belonging to special output plugins. For this you need to know the name of the output plugin you want to observe a parameter and the name of the parameter.

Manipulate output plugins directly:

- See section Interface and how to use for various actions possible on output plugin pointers that you got by calling `get_default_output` or creating new ones (see section Access Output Plugins).

Observe parameters by *calling GAV user API functions*:

- Adding a parameter for observation to a default or manually created output plugin can be done using the function `OutputPlugin_if* add_to_output(OutputPlugin_if*, gs_param_base*)`. The output plugin (first parameter) is one of the pointers returned by the functions above (`get_default_output` and `create_outputPlugin`).
- Another way observing a parameter with one of the default plugins is calling `OutputPlugin_if * add_to_default_output(OutputPluginType, gs_param_base*)` which adds a parameter to the default instance of the specified type. It returns a pointer to the default output plugin, too.
- The easiest way observing a parameter with the simulation-wide default output is calling the function `OutputPlugin_if* add_to_default_output(gs_param_base*)`. It returns a pointer to the default output plugin, too.

Observe parameters by *creating and manipulating parameters belonging to special output plugins*:

- To enable the observation of a parameter (which may be already existing or may not), create a new implicit parameter which is a child parameter of the output plugin and is named as the parameter that should be observed. This comes clear in the following example:
  - The GAV Plugin is named "AnalysisPlugin".
  - The output plugin which should observe the parameter is the default stdout one, automatically named "STDOUT_OUT_default".
  - The parameter that should be observed is named "mymodule.submodule.myparam".
  - To observe the parameter with the output plugin, simply create a so-called *enabled parameter* (child of the output plugin) named "AnalysisPlugin.STDOUT_OUT_default.mymodule.submodule.myparam" by doing the following call on the config user API:

    ```
    mCnfApi->setInitValue("AnalysisPlugin.STDOUT_OUT_default.mymodule
        .submodule.myparam", "true");
    ```

    The output plugin will automatically make the implicit parameter explicit immediately.
  - Afterward you can enable and disable the observation of the parameter by modifying the 'enabled parameter' to true or false, e.g.

    ```
    gs::gs_param<bool> *enabled
      = mCnfApi->get_gs_param<bool>("AnalysisPlugin.
          STDOUT_OUT_default.mymodule.submodule.myparam");
    *enabled = false;
    ```

This works as well with any manually created output plugin. E.g. to add a parameter to an output plugin named “mymodule.main_action.STDOUT_OUT_userChosenConstructorParam” this is done by calling

```
mCnfApi->setInitValue("mymodule.main_action.
    STDOUT_OUT_userChosenConstructorParam.mymodule.param2", "true");
```

Output Plugin Names

An output plugin is an sc_object which is named in the following pattern. The name is needed e.g. when creating ’enabled parameters’ (see above) for this output plugin.

All output plugin names have their name string as prefix (e.g. STDOUT_OUT) and their constructor parameter as tail.

The default output plugins (one for each available output plugin type) are created and owned by the GAV Plugin. Their name tails are _default (which is the constructor parameter specified by the GAV Plugin).

The tail is automatically converted to SystemC conform names by replacing each character which is not a letter or a digit or an underscore to an underscore.

Examples:

- *Default* output plugin of type STDOUT_OUT:
  name: AnalysisPlugin.STDOUT_OUT_default
- *Default* output plugin of type TXT_FILE_OUT:
  name: AnalysisPlugin.TXT_FILE_OUT_default
- User created output plugin created *in the constructor* of module mymodule with the call `create_OutputPlugin(TXT_FILE_OUT, "myop")`:
  name: mymodule.TXT_FILE_OUT_myop
- User created output plugin created *in the constructor* of module mymodule with the call `create_OutputPlugin(TXT_FILE_OUT, "myop.txt")`:
  name: mymodule.TXT_FILE_OUT_myop_txt
- User created output plugin created *in the thread* main_action of module mymodule with the call `create_OutputPlugin(TXT_FILE_OUT, "myop.txt")`:
  name: mymodule,main_action.TXT_FILE_OUT_myop_txt

Interface and how to use

After the user got an output plugin pointer of type `OutputPlugin_if` (see OutputPlugin_if) it may be used: Call

- `observe(gs_param_base&)` to let the output plugin observe a parameter.
- `observe_all(GCnf_Api& config_api)` to observe all parameters currently existing.

- `observe(vector<gs_param_base*>)` to observe parameters given in a vector (e.g. as a result of the config user API call `getParams(string)`).
- `observe(string getParamList_string)` to observe parameters specified by a string like "myModule.*". Internally the config API call `getParams(string)` will be used. To observe all parameters underneath the hierarchy level of myModule, call `observe("myModule.*")`.
- `remove(gs_param_base&)` to remove a parameter observation from the output.
- `pause()` to pause the output from now on.
- `pause(sc_event&)` to pause the output until the given event is notified.
- `pause(sc_time&)` or `pause(double, sc_time_unit)` to pause the output for the given time.
- `resume()` to resume the output manually if paused.

Consult the doxygen API [1] for more details.

An output plugin is not paused by default.

The usage of output plugins is shown in the example 📁 greencontrol/examples/gav_simple , especially in the files 🗋 AVAnalyserTool.* .

The type of output plugins is defined with an enum OutputPluginType (see file 🗋 gav_datatypes.h).

[1] GreenAV doxygen API: `http://www.greensocs.com/projects/GreenControl/GreenAV/docs/GreenAVDoxygen` (take care to look at the correct release)

Code example Listing 2.3 and figure 2.4 show examples how to use output plugins.

```
// get pointer to parameter bases
gs::gs_param_base *int_par  = m_configAPI.getPar("Owner.int_param");
gs::gs_param_base *str_par  = m_configAPI.getPar("Owner.str_param");
gs::gs_param_base *uint_par = m_configAPI.getPar("Owner.uint_param");

// get simulation-wide default output, and observe a param
gs::av::OutputPlugin_if* overallDefaultOP = m_analysisAPI.↩
   get_default_output();
m_gavApi.add_to_output(overallDefaultOP, int_par);
// alternative: add params to the default text file output
gs::av::OutputPlugin_if* fileOP =
    m_gavApi.add_to_default_output(gs::av::TXT_FILE_OUT, int_par);
m_gavApi.add_to_default_output(gs::av::TXT_FILE_OUT, uint_par);

// create an additional CSV-file output plugin and add all existing
// parameter to output
gs::av::OutputPlugin_if* csvFileOP =
    m_gavApi.create_OutputPlugin(gs::av::CSV_FILE_OUT, "CSVexp.log");
csvFileOP->observe_all(m_configAPI);

// pause for 2 ns.
csvFileOP->pause(2, SC_NS);

// remove parameter from default file output
fileOP->remove(*int_par);
```

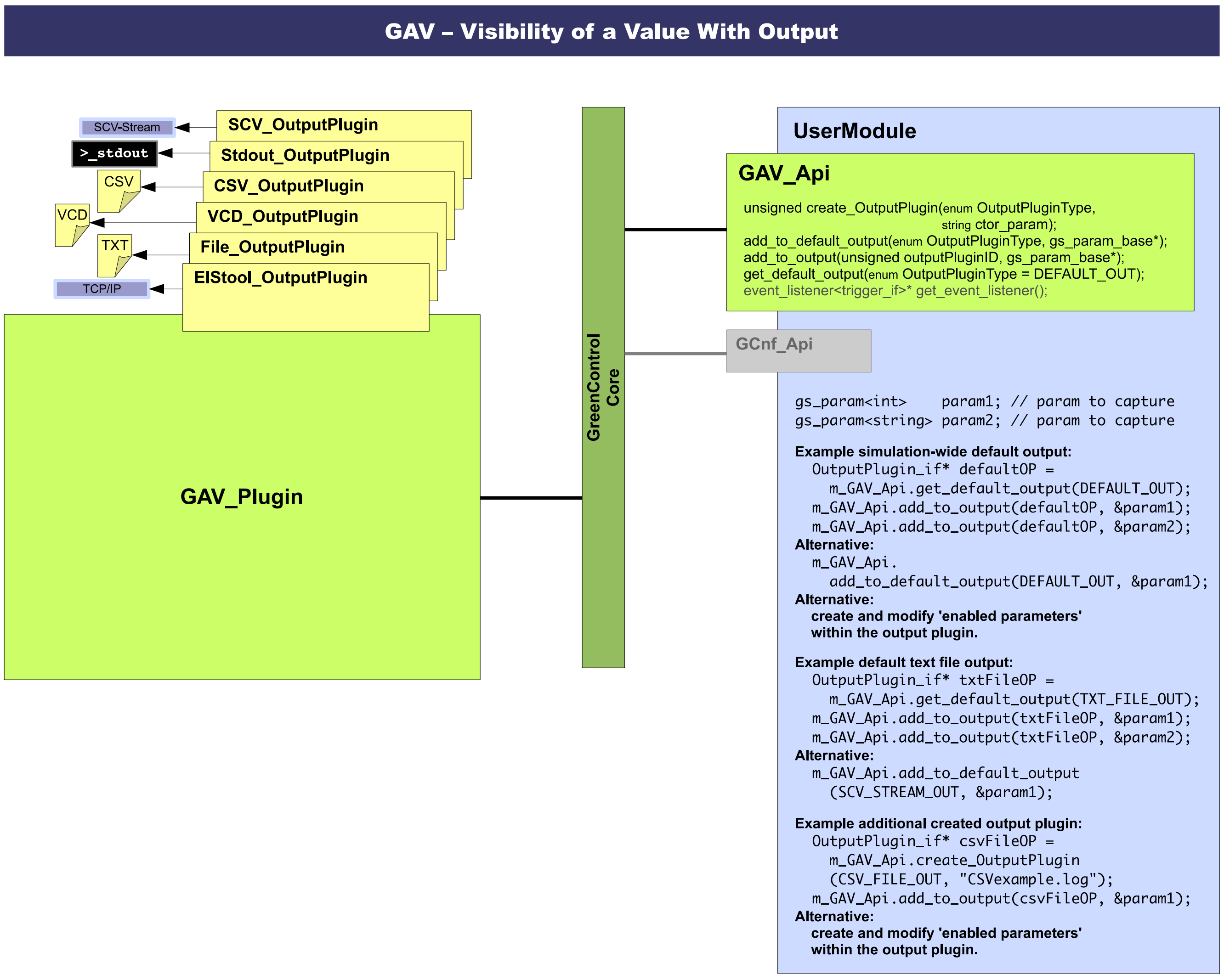
Listing 2.3: Output plugin code example

Implementation Note

Output Plugin

The output plugin instances are newed by the GAV plugin and stored in two maps: one for the default (`m_DefaultOutputPlugins`) and one for the additional (`m_OutputPlugins`) output plugins.

One event listener (`m_OutpPl_event_listener`, see section 2.5.3) is owned by the GAV plugin to handle events (pause and resume) of output plugins.



```
unsigned create_OutputPlugin(enum OutputPluginType,
                                          string ctor_param);
add_to_default_output(enum OutputPluginType, gs_param_base*);
add_to_output(unsigned outputPluginID, gs_param_base*);
get_default_output(enum OutputPluginType = DEFAULT_OUT);
event_listener<trigger_if>* get_event_listener();
```

```
gs_param<int>    param1; // param to capture
gs_param<string> param2; // param to capture
```

Example simulation-wide default output:

```
  OutputPlugin_if* defaultOP =
    m_GAV_Api.get_default_output(DEFAULT_OUT);
  m_GAV_Api.add_to_output(defaultOP, &param1);
  m_GAV_Api.add_to_output(defaultOP, &param2);
```

Alternative:

```
  m_GAV_Api.
    add_to_default_output(DEFAULT_OUT, &param1);
```

Alternative:
create and modify 'enabled parameters' within the output plugin.

Example default text file output:

```
  OutputPlugin_if* txtFileOP =
    m_GAV_Api.get_default_output(TXT_FILE_OUT);
  m_GAV_Api.add_to_output(txtFileOP, &param1);
  m_GAV_Api.add_to_output(txtFileOP, &param2);
```

Alternative:

```
  m_GAV_Api.add_to_default_output
    (SCV_STREAM_OUT, &param1);
```

Example additional created output plugin:

```
  OutputPlugin_if* csvFileOP =
    m_GAV_Api.create_OutputPlugin
    (CSV_FILE_OUT, "CSVexample.log");
  m_GAV_Api.add_to_output(csvFileOP, &param1);
```

Alternative:
create and modify 'enabled parameters' within the output plugin.

Figure 2.4: GreenAV Output Example

Implementation Note

OutputPlugin_base

An output plugin derives from the base class `OutputPlugin_base` (see file OutputPlugin_base).

Each output plugin needs only implement its constructor, destructor, the initialization function `init()` and the callback function `config_callback` which should perform the actual output.
Output plugins implement the constructor which gets one string parameter to allow adding the output plugin to the GAV plugin's create command.
All functions are virtual to give the derived class the oportunity to catch the calls before calling the base class' function.

The derived classes need to call `init()` on the first access (dependent on the variable `bool is_used`), not during construction. The constructor should not do memory allocations etc. because it is called (for the default output plugins) even if the output plugin will never be used.

2.4.1 Default Output Plugin (special case)

- Identify the default output plugin (which is actually one of the following) with gs::av::DEFAULT_OUT,
  id (OutputPluginType) = 0.

2.4.2 NULL Output Plugin (special case)

- Identify the NULL output plugin (which is actually no one) with gs::av::NULL_OUT,
  id (OutputPluginType) = 1.

Use this setting if the output should go nowhere.

2.4.3 STDOUT Output Plugin

- Included within greencontrol/analysis.h
- Implementation see file Stdout_OutputPlugin.h
- Identify this output plugin with gs::av::STDOUT_OUT,
  id (OutputPluginType) = 3.

The STDOUT output plugin can be used to simply display parameter changes to the standard output (e.g. terminal where the simulation runs).

One line per parameter change is printed and contains timing (and delta) information, the parameter name and the new value. An output example can be seen in listing 2.4.

This output plugin is a very convenient way to print each parameter change to the simulation output.

The constructor string parameter is not used within this output plugin.

```
@130 ns /7 (Stdout_OutputPlugin): AVnewStatCalc.int_par = 2
```

Listing 2.4: STDOUT output example

2.4.4 Text-file Output Plugin

- Include greencontrol/analysis_file_outputplugin.h
- Implementation see file File_OutputPlugin
- Identify this output plugin with gs::av::TXT_FILE_OUT,
  id (OutputPluginType) = 2.

The Text-file output plugin can be used to record parameter changes to a human-readable text file. The format is similar to the STDOUT plugin's output: The first line is the simulation time, the following lines contain one line per parameter change. See listing 2.5 for an example.

The constructor parameter is the filename of the output file. Be careful to set the file extension (e.g. .txt or .log). If the given file name has no extension (no dot in it) the extension .log will be added automatically.

```
Simulation time: Fri Mar 28 15:03:09 2008

@1 ns /1: Owner.int_param = 100
@1 ns /1: Owner.uint_param = 670
@1 ns /1: Owner.int_param = 101
@2 ns /3: Owner.int_param = 102
```

Listing 2.5: Text-file output example

Pure Output Pure output is a special feature of the text-file output plugin. The additional function `void pure_output(const std::string&)` can be called to write directly to the text file without waiting for parameter change callbacks and without using the formating functionality of the output plugin. This feature is used by report message streamers (see project *ReportMessages*).

```
std::string mystring;
mystring = "Any string to output to file"; mystring += std::endl;
gs::av::OutputPlugin_if* op
  = m_analysisAPI->create_OutputPlugin(gs::av::TXT_FILE_OUT, "file.log");
gs::av::File_OutputPlugin* fop
  = dynamic_cast<gs::av::File_OutputPlugin*>(op);
fop->pure_output(mystring);
```

Listing 2.6: Text-file *pure* output example

2.4.5 Comma Separated Values (CSV) Output Plugin

- Include greencontrol/analysis_csv_outputplugin.h
- Implementation see file CSV_OutputPlugin
- Identify this output plugin with gs::av::CSV_FILE_OUT, id (OutputPluginType) = 4.

The Comma Separated Values (CSV) output plugin can be used to output parameter changes that should be imported to MS Excel.

The constructor parameter of the `CSV_OutputPlugin` is the filename of the output file. The default filename (for the plugin's default output plugin) is default.csv. If the filename string has not the postfix '.csv' this will be appended automatically.

Due to the structure of CSV files all parameters must be registered before writing the file! The plugin will begin writing the file on the first parameter change.

The behavior how to react to newly added parameters after began writing can be adapted with the `#define ALLOW_ADDING_PARAMETERS_AFTER_HEADER_WRITTEN`. If not defined the output plugin will refuse (ignore) all register requests for new parameters after having written the first value! If defined the output plugin will add this new observed parameter to each following line. Accordingly the row witdh may increase within the CSV-file - which is imported by Excel without problems. But the head of the table will *not* include the parameter name, the last columns will remain unnamed!

Pausing the plugin does *not* write the current time slot to the file because the plugin may be resumed again during the same time slot.

The separator character can be specified with the macro `#define SEPARATOR ';'`. For standard-conform CSV-files this separator has to be a `','`, for files to be imported by Excel `';'` should be used (default).

The following text file is an example for a file created by this plugin. The first line is the time of the simulation run, the thirs line is the output name (constructor parameter, file name), the fifth line is the header of the table containing the parameter names. All remaining lines are data (parameter changes). This example includes integer and string parameters. Figure 2.5 shows the file opened with Excel.

```
Simulation time: Fri Mar 28 14:00:42 2008

CSVexample.log

"time /delta";"Owner.int_param";"Owner.str_param";"Owner.uint_param";
"1 ns /1";"101";"Hello World!";"670"
"2 ns /3";"104";;
"2 ns /4";"106";"Hello Germany!";
"2 ns /5";;"Hello Arizona!";
"2 ns /6";;"Hello France!";
"3 ns /7";;;"2000"
"5 ns /8";"222";"new hello";"3000"
"106 ns /10";"133";;
"210 ns /18";"10000";;
```

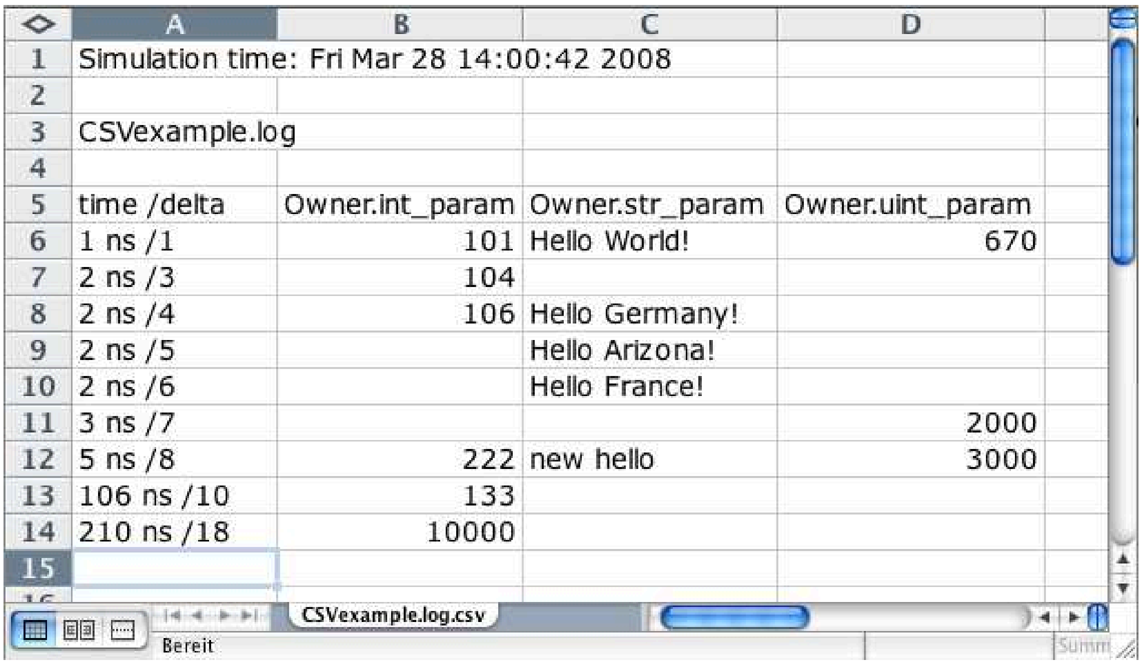
Listing 2.7: CSV output file example

2.4.6 SCV-stream Output Plugin

- Include greencontrol/analysis_scv_outputplugin.h
- Implementation see file SCV_OutputPlugin
- Identify this output plugin with gs::av::SCV_STREAM_OUT, id (OutputPluginType) = 5.

The SCV output plugin (`SCV_OutputPlugin`) outputs parameter changes to a stream of the SystemC Verification Standard (SCV). This standard stream can be used as an input to several vendor tools (such as CoWare and Mentor).

This output plugin exports the registered parameter changes to an *SCV stream* named GreenAVstream_<OutputPluginName>. The constructor string parameter is the OutputPluginName.



	A	B	C	D
1	Simulation time: Fri Mar 28 14:00:42 2008			
2				
3	CSVexample.log			
4				
5	time /delta	Owner.int_param	Owner.str_param	Owner.uint_param
6	1 ns /1	101	Hello World!	670
7	2 ns /3	104		
8	2 ns /4	106	Hello Germany!	
9	2 ns /5		Hello Arizona!	
10	2 ns /6		Hello France!	
11	3 ns /7			2000
12	5 ns /8	222	new hello	3000
13	106 ns /10	133		
14	210 ns /18	10000		
15				

Figure 2.5: CSV ouput plugin Excel example

Each parameter gets its own transaction generator. When a parameter changes, the previous transaction is ended and a new one with the new value is started. Accordingly a transaction represents the time where the value of a parameter remains unchanged.

The database that should be written to can be chosen in the static function `init_scv_recording()` (see file SCV_OutputPlugin.h).

The output plugin's *default database* is a text file which is the default database in the SCV framework. The SCV stream output is written to the file transaction_text_db . Add #define areas to use other databases (like ModelSim).

There are some options to change the output plugin's behavior to be able to support a wide range of external tools reading the SCV stream. These options can be changed in the define area at the top of the file SCV_OutputPlugin.h .

- The default is to use only one globals database for all SCV output plugin instances. If there is a database type that is able to handle more than one database, undefine the `#define ONLY_ONE_GLOBAL_DATABASE`.
- If a tool or database is only able to handle one stream, use the `#define ONLY_ONE_GLOBAL_STREAM` (defined by default).
  Then all `SCV_OutputPlugin` instances write to the same stream.
- The OutputPlugin is able to create transactions with values of the type string (default) The user may set `#define USE_CORRECT_TYPE_TRANSACTIONS true` to enable transactions containing values of the correct type (as long as the type is listed in the enum gs::cnf::Param_type and is implemented here). This is slower because of some additional switch statements and casts.

The files AVscvAnalyserTool.* in the example greencontrol/examples/gav_simple create SCV output streams.

Experience with Mentor Graphics ModelSim

ModelSim (Mentor Graphics) has the ability to run SystemC simulations and to show the results graphically. GreenBus and GREENCONTROL (including GREENCONFIG and GREENAV) can be compiled with Mentor's sccom compiler and can be run with the vsim simulator.

Mentor's SCV implementation does not support string data types in transactions being recorded in an SCV stream. Accordingly the SCV output plugin has to use the USE_CORRECT_TYPE_TRANSACTIONS define and cannot show string parameters. So the visualization is restricted to the data types listed in the enumeration gs::cnf::Param_type and implemented in the plugin.

The SCV output plugin has an `init_recording` function which automatically uses ModelSim special code when compiled with the ModelSim sccom compiler. In the define section there is an `ifdef` for ModelSim which automatically sets the correct behavior.

These steps have to be performed when simulating with ModelSim:

- Start the simulation (*Important!* The transaction stream does not occur before simulation is running and has created the stream!)
- View the recorded SCV transactions by right clicking on the module which contains the stream and selecting Add $\longrightarrow$ Add All Signals to Wave.

Experience with CoWare

Experiments with the CoWare Platform Architect:

To use the CoWare settings, #define `CoWare`. The SCV output plugin has a special define area in its `init_recording` function for the CoWare tool. In the define section there is an `ifdef` for CoWare which automatically sets the correct behavior.

2.4.7 How to implement an Output Plugin

This is a short introduction how to implement a new output plugin. For a simple example see the STDOUT output plugin in file Stdout_OutputPlugin.h (see section 2.4.3).

- Outside the class (but in same namespace) specify the project-wide unique id and convenience name with the macro GAV_REGISTER_PLUGIN(unique_id, convenience_variable, class_name). Example:

```
GAV_REGISTER_PLUGIN(10, MY_NEW_OUT, MyNew_OutputPlugin);
```

- Inherit `OuputPlugin_base` which provides most functionality expect the output itself.
- Do not implement the empty constructor.

- Implement the constructor taking a name and an event listener and forward the event listener to the base class:

```
MyNew_OutputPlugin ( const char* unused,
                     event_listener<OutputPlugin_base> *ev_listn)
  : OutputPlugin_base(ev_listn)
```

- Implement the callback function and make sure to remove a parameter being observed by the output plugin when is is being destroyed. Guard the output by checking `is_running`.

```
void config_callback(gs_param_base &par) {
  if (par.is_destructing()) {
    remove(par);
  }  else if (is_running) {
    // Do the output
  }
}
```

- Implement the output within the callback function (previous bullet point). Use the functions the `gs_param_base` provides or cast to the actual `gs_param<type>` type (see GreenConfig User's Guide[2]).

```
par.getName();
par.getString();
```

- Create a member representing the new output plugin in the enum OutputPluginType in file gav_datatypes.h .

```
enum OutputPluginType {
  [... other members ...],
  // Short explanation of my outp.pl.
  MYNEW_OUTPUT_PLUGIN
}
```

- Add the new output plugin to the GAV plugin's `outputPluginFabricCreator` function (file gav_plugin.h) by adding a switch case. The case is the enum member being created at the preceding bullet point.

```
case MYNEW_OUTPUT_PLUGIN:
{
  // create MyNew output plugin
  op = new MyNew_OutputPlugin(constructParam, &↩
     m_OutpPl_event_listener);
  break;
}
```

[2]GreenConfig User's Guide: http://www.greensocs.com/projects/GreenControl/GreenConfig

Advanced Note
Write an Output Plugin from scratch
When writing a completely new output plugin at least the interface OuputPlugin_if has to be implemented. The constructor should get (but needs not) the string given by the user, transmitted over the transaction.

2.5 Analysis and Visibility Service

2.5.1 Service Plugin

The service plugin (class `GAV_Plugin`, file gav_plugin.h) uses the enum member AV_SERVICE and the port plugin name GAV_Plugin.

The main functionality of the plugin:

- Manage output plugins (see section 2.4),
- Own two event listeners needed by output plugins and Statistics Calculator Triggers (see sections 2.4 and 2.6).

2.5.2 Commands

The analysis and visibility service AV_SERVICE uses the command enum GAVCommand (see file gav_datatypes.h).

These commands may be used for the API – Plugin communication:

Direction:API $\longrightarrow$ GAV Plugin	
Command	Usage
`CMD_ADD_TO_OUTPUT_PLUGIN`	Adds a parameter to an output plugin. The output plugin can be either the default one of the specified type or a user API specified one. If the default one is being used for the first time the output plugin will be instantiated by the plugin.
`CMD_CREATE_OUTPUT_PLUGIN`	Creates a new output plugin of the specified type and returns the pointer to the API.
`CMD_GET_EVENT_LISTENER`	Returns the pointer to the plugin's trigger event listener to the API.

The fields of the transaction are used for special commands on this way:

Command	Phase	Field	Description
`CMD_ADD_TO_OUTPUT_PLUGIN`	REQUEST	*AnyPointer*	$!= NULL$: 'Identifier' (pointer) of the output plugin the parameter should be added to. $== NULL$: add to the default output plugin.
	REQUEST	*AnyUint*	If ($AnyPointer == NULL$): type of the default output plugin (member of enum OutputPluginType).
	REQUEST	*AnyPointer2*	Optional: Pointer (void*) to the parameter that should be added. $== NULL$: and if $AnyPointer == NULL$: only return the default output plugin pointer without observing a parameter
	RESPONSE	*AnyPointer*	Pointer ('identifier') to the output plugin the parameter was added. Can be used in this command for adding further parameters. $= NULL$: on error.
	RESPONSE	*Error*	> 0 when adding fails.

Command	Phase	Field	Description
`CMD_CREATE_OUTPUT_PLUGIN`	REQUEST	*AnyUint*	Type of the OutputPlugin that should be created (member of enum OutputPluginType).
	REQUEST	*Value*	Constructor parameter (string) that should be given to the output plugin's constructor (containing e.g. filename dependent on the output plugin type).
	RESPONSE	*AnyPointer*	Pointer ('identifier') to the created output plugin. Can be used in the CMD_ADD_TO_OUTPUT_PLUGIN command for adding parameters.
	RESPONSE	*Error*	No error specified so long.

Command	Phase	Field	Description
`CMD_GET_EVENT_LISTENER`	RESPONSE	*AnyPointer*	Pointer to the plugin's trigger event listener.
	RESPONSE	*Error*	No error specified so long.

2.5.3 Event Listener

Some analysis objects have to wait for events. These objects may not be allowed to be an sc_module because they may be created during simulation runtime (e.g. the trigger of the statistics calculator, see section 2.6.1). To be able to wait for `sc_events` anyway, that object may use *event listeners*.

Event listeners are sc_modules owned by the plugin doing dynamically spawned event waits: An event listener is is templated to the initiator who wants to wait for an event. When the initiator calls `create_event_listener` the listener dynamically spawns a process being sensitive for the given event. When the event is notified the initiator is called back by the listener process.

The plugin has two event listeners:
`event_listener<trigger_if>` and `event_listener<OutputPlugin_base>`. The one for triggers can be accessed using the GAV user API by calling `get_event_listener` (see section 2.3).

The listener pointer templated to `trigger_if` can be transported to the API via a transaction of command type CMD_GET_EVENT_LISTENER. The listener templated to `OutputPlugin_base` is used by the plugin itself when instantiating output plugins.

2.6 Statistics Calculator

Concepts

Most analysis tasks are performed by the *Statistics Calculator* (StatCalc) which is realized in the class `StatCalc` in file StatCalc.h . The Statistics Calculator (class `StatCalc`) manages the analysis of several input parameters that are calculated in a formula on several activation events. The tasks and the corresponding interfaces can be classified as follows:

1. **Calculation Interface** (Calculator)
   - Math operations
   - Logical operations for boolean expressions usable as activation event
2. **Calculation Activation Events** (Trigger)
   - Per default the result is updated (re-calculated) each time one of the *formula (input) parameter* changes.
   - A boolean condition can guard the calculation.
     The calculation will only be performed if the given parameter is true. Note: A change of the condition parameter from false to true does not trigger re-calculation.
   - A (complex) boolean expression can be created using another Statistics Calculator object giving its bool result parameter to `set_condition(gs_param<bool>&)`.
   - An sc_event can be used to trigger the calculation. Each time the event is notified the calculation will be performed. (If there is a boolean condition activated this will be checked first.)
   - A fixed interval, which triggers re-calculation each given time, can be specified.
   - The calculation can also be manually performed by a simple function call.
3. **Statistics Abilities** (Calculator)
   - *Sliding window*: The user can insert a sliding window between the calculation and the result.
     The sliding window puts out the average over the last *n* values to the result parameter.
     All activation mechanisms keep the same (parameter change callbacks, boolean conditions, sc_event, interval).

Figure 2.6 shows the *Statistics Calculator concept*. The StatCalc consists mainly of a trigger object (see section 2.6.1) and a calculator object (see section 2.6.2). The calculator is responsible for managing the formula that should be calculated. The trigger manages the activation events of re-calculation and then calls the calculator to perform re-calculation.

The StatCalc object gets a trigger and a calculator object during construction and combines them. It gives the calculator functor object to the trigger object to enable the trigger to activate the recalculation. The StatCalc calls `get_used_params()` on the calculator to give all input parameters to the trigger (using `set_used_params()`.

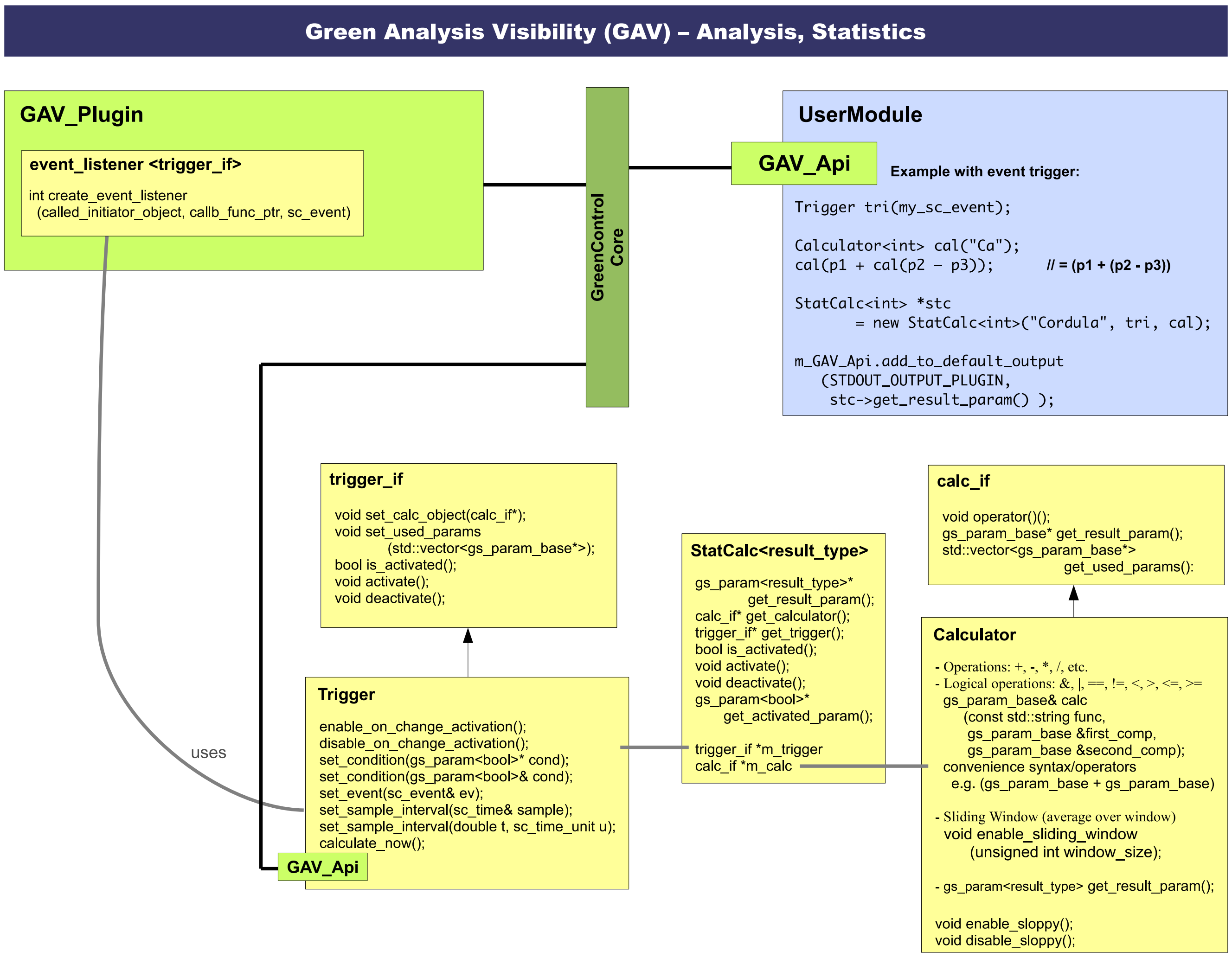
Interface and how to use

The `StatCalc` object has to be instantiated *after* the trigger and calculator have been created and specified completely.

The *name* of the StatCalc has to be specified during construction and is made unique by the constructor using `sc_gen_unique_name`.

The full constructors are:

```
StatCalc(const char* stcalc_name,
                   trigger_if* any_trigger, calc_if* any_calculator);
StatCalc(const char* stcalc_name,
                   trigger_if& any_trigger, calc_if& any_calculator);
```


Green Analysis Visibility (GAV) – Analysis, Statistics
GAV_Plugin
event_listener <trigger_if>
int create_event_listener
(called_initiator_object, callb_func_ptr, sc_event)
GreenControl Core
UserModule
GAV_Api
Example with event trigger:
uses
trigger_if
void set_calc_object(calc_if*);
void set_used_params
(std::vector<gs_param_base*>);
bool is_activated();
void activate();
void deactivate();
Trigger
enable_on_change_activation();
disable_on_change_activation();
set_condition(gs_param<bool>* cond);
set_condition(gs_param<bool>& cond);
set_event(sc_event& ev);
set_sample_interval(sc_time& sample);
set_sample_interval(double t, sc_time_unit u);
calculate_now();
GAV_Api
StatCalc<result_type>
gs_param<result_type>*
get_result_param();
calc_if* get_calculator();
trigger_if* get_trigger();
bool is_activated();
void activate();
void deactivate();
gs_param<bool>*
get_activated_param();
trigger_if *m_trigger
calc_if *m_calc
calc_if
void operator()();
gs_param_base* get_result_param();
std::vector<gs_param_base*>
get_used_params():
Calculator
- Operations: +, -, *, /, etc.
- Logical operations: &, |, ==, !=, <, >, <=, >=
gs_param_base& calc
(const std::string func,
gs_param_base &first_comp,
gs_param_base &second_comp);
convenience syntax/operators
e.g. (gs_param_base + gs_param_base)
- Sliding Window (average over window)
void enable_sliding_window
(unsigned int window_size);
- gs_param<result_type> get_result_param();
void enable_sloppy();
void disable_sloppy();

```
Trigger tri(my_sc_event);

Calculator<int> cal("Ca");
cal(p1 + cal(p2 - p3));       // = (p1 + (p2 - p3))

StatCalc<int> *stc
       = new StatCalc<int>("Cordula", tri, cal);

m_GAV_Api.add_to_default_output
   (STDOUT_OUTPUT_PLUGIN,
    stc->get_result_param() );
```

Figure 2.6: GreenAV Statistics Calculator Concept

Alternative constructors do not need a trigger to be specified. In this case the default trigger object will we created by the StatCalc. The default trigger triggers on each input parameter change.

```
StatCalc(const char* stcalc_name, calc_if* any_calculator);
StatCalc(const char* stcalc_name, calc_if& any_calculator);
```

The StatCalc object gives access to the trigger and calculator object pointers via the functions `trigger_if* get_trigger()` and `calc_if* get_calculator()` and provides some of the trigger and calculator interface functions whose calls will be redirected to the trigger object:

```
gs_param<result_type>* get_result_param(); // redirected to calculator
calc_if* get_calculator(); // local
trigger_if* get_trigger(); // local
bool is_activated(); // redirected to trigger
void activate();     // redirected to trigger
void deactivate();   // redirected to trigger
```

The activation status of the trigger can also be accessed and manipulated with the parameter <statcalc_name>.activated which is returned by the function:

```
gs_param<bool>* get_activated_param(); // local
```

2.6.1 Trigger

Concept

The trigger is responsible for activation events. When one event occurs, the re-calculation of the calculator functor has to be called so the trigger object triggers the calculation.

The trigger is an object that must inherit `trigger_if` to allow it to be managed by the StatCalc object.

Interface

All trigger classes have to implement the virtual interface `trigger_if`. The following listing lists the functions to be implemented by a trigger class:

```
class trigger_if {
  virtual ~trigger_if() { }
  // Initialization functions
  virtual void set_activated_param(gs::gs_param<bool> &activated) = 0;
  virtual void set_calc_object(calc_if*) = 0;
  virtual void set_used_params(std::vector<gs_param_base*>) = 0;
  // User functions
  virtual bool is_activated() = 0;
  virtual void activate() = 0;
  virtual void deactivate() = 0;
  // Callback functions
  virtual void event_callback() = 0;
  virtual void interval_callback() = 0;
};
```

Listing 2.8: Trigger interface trigger_if

For a detailed description see the API reference.

The initialization functions are called by the StatCalc during its construction.

The function `set_activated_param` gets a reference to the parameter that has been created by the StatCalc and which represents and modifies the activation status. The trigger has to register a callback for this parameter to react on changes. This function is called before the other initialization functions to ensure that the parameter is available.

The function `set_used_params` gets the formula parameters of the calculator. These parameters have to be observed by the trigger for destruction. After one of these parameters has been destructed the trigger must never again call the re-calculation of the calculator!

The function `deactivate` should deactivate the trigger until the function `activate` is called. A deactivated trigger should unregister all input parameter callbacks (for changes and destruction checks) due to performance reasons: A deactivated trigger should produce no overhead! This allows creation of (deactivated) Statistics Calculators without runtime overhead. Get if the trigger is activated by calling `is_activated`.

The functions `event_callback` and `interval_callback` are both functions that can be registered by the trigger to the event listener of the GAV plugin (which may be accessed via the `GAV_Api`) which is templated to `trigger_if`.

Advanced Note

Re-implement a trigger

As an extension to the GreenAV framework new triggers can be implemented. They just have to implement the `trigger_if` interface.
This may be needed if further activation events are needed that are not met by the default GreenAV trigger.
When re-implementing a trigger be careful never calling the re-calculation if one of the formula parameters has been destroyed. Also take care to deactivate the trigger completely when the deactivation function is being called. On activation you have to check whether all formula parameters are still existing.

GreenAV default Trigger and calculation activation events

The default *GreenAV trigger* object class is `Trigger`.

The trigger automatically registers for callbacks of all formula (input) parameters which are given to the trigger via the `set_used_params` call. The trigger handles destructed input parameters by deactivating the trigger forever and never again calling the calculator functor object. Even if parameter changes are not a calculation activating event, the trigger registers for callbacks just to check if the changed parameter is being destructed.

Calculation activation event kinds this Trigger supports are:

- sc_event for *SystemC events*:

  The sc_event calculation activation event uses the trigger event listener of the GAV plugin to wait for the user-given event. When the event is notified the trigger will be called back (function `event_callback`) by the event trigger.

- sc_time for *periodical sample intervals* (periodically repeated events)

  The sc_time calculation activation event notifies the trigger itself periodically after the specified time intervals. This trigger also uses the API's event listener. This may be used e.g. for sliding windows that record on fixed time intervals.

- *value changes* on input parameters (uses parameter callbacks)

  This is the *default* trigger mechanism. The trigger registers callbacks for all formula input parameters and triggers re-calculation an each callback.

- *manually chosen parameter changes* The user may give any gs_params to the trigger whose value changes will result in a re-calculation. These parameters need not (but may) be formula parameters.

- *manually*

  The user may manually trigger the re-calculation with a simple function call.

Additionally the Trigger supports a

- *Guard* (*no* activation event): `gs_param<bool>` for a condition to be checked before calculation.

  This guard parameter is checked each time the trigger got one of the activation events. If the guard's value is true the re-calculation is performed, otherwise re-calculation is suppressed.

The trigger is able to handle several activation events concurrently. The user may enable value change activation and e.g. set one (or more) sc_event(s) on whose notification triggering will also be performed. Manually triggering is always possible (taking account of the guard as well).

Implementation Note

Event listener

The trigger needs the API's event listener templated to `trigger_if`. During construction the trigger gets a `GAV_Api` instance and calls `get_event_listener` and stores the pointer.

Afterwards the trigger calls `m_event_listener->create_event_listener(this, & trigger_if::event_callback, ev);` where it is needed to create a dynamically spawned process that waits for the event ev.

How to use

This section is about how to prepare the trigger in the user's code before giving it to the `StatCalc` object.

The specific constructors specify the way the trigger waits for calculation activation events. The vector parameter takes manually chosen parameters. There are constructors for many combinations of activation events:

```
explicit Trigger();
explicit Trigger(sc_event& event);
explicit Trigger(sc_event* event);
explicit Trigger(sc_time ti);
explicit Trigger(double t, sc_time_unit u);
explicit Trigger(bool on_changes);
explicit Trigger(sc_event& event, gs_param<bool> &cond);
explicit Trigger(sc_time &ti, gs_param<bool> &cond);
explicit Trigger(double t, sc_time_unit u, gs_param<bool> &cond);
explicit Trigger(bool on_changes, gs_param<bool> &cond);
explicit Trigger(gs_param<bool> &cond);
explicit Trigger(gs_param<bool> *cond);
explicit Trigger(sc_event &ev, sc_time &ti, bool on_changes,
                 gs_param<bool> cond);
explicit Trigger(sc_event &ev, sc_time &ti, gs_param<bool> &cond);
explicit Trigger(sc_event &ev, sc_time &ti);
explicit Trigger(sc_event &ev, bool on_changes);
explicit Trigger(sc_time &ti, bool on_changes, gs_param<bool> &cond);
explicit Trigger(sc_time &ti, bool on_changes);
explicit Trigger(std::vector<gs_param_base*>);
```

Listing 2.9: Trigger constructors with several calculation activation events and guard

The constructor activation event and guard order is: sc_event, sc_time, bool_on_changes, guard.

There may be activation event combinations which are not available, e.g. when combining manually chosen parameters with other event.Then the user may use one constructor and one of the following functions after construction:

- `set_event(sc_event& ev)` enables trigger for SystemC events.

  When called repeatedly all events are valid. Removing events is not possible.

- `set_sample_interval(sc_time& sample)` enables periodical sample interval trigger.

When called repeatedly the last interval is the valid one.

- `enable_on_change_activation()` enables trigger for value changes,

  `disable_on_change_activation()` may be used to disable the value change trigger while the trigger 'runs'.

- `set_condition(gs_param<bool>* cond)` or `set_condition(gs_param<bool>& cond)` sets the guard parameter.

  `cond = NULL` removes the guard.
  When being called repeatedly the old guard will be overwritten.

All these functions may be called anytime (even if trigger already 'runs').

Further available functions are:

- `calculate_now()` triggers the re-calculation manually. This call takes account of the activation and the guard just like all activation events do.
- `bool is_activated()` returns if the trigger / StatCalc is activated, see interface description above.
- `void activate()` activates the trigger / StatCalc, see interface description above.
- `void deactivate()` deactivates the trigger / StatCalc, see interface description above.
- `void set_activation_status(bool _activate)` calls `activate` or `deactivate`.

The following listings are some usage examples:

For the example this predefinitions are needed:

```
#include "greencontrol/analysis.h"
```

```
gs::av::Trigger *stc_tom_t = new Trigger(true);
  // is the same as
gs::av::Trigger *stc_tom_t = new Trigger();
```

Listing 2.10: Example: standard value change trigger.

```
gs::gs_param<bool> condition("condition");
gs::av::Trigger stc_regina_t(condition); // uses param changes and a guard
'... create calculator, StatCalc ...'
'... do something ...'
// disable standard value change trigger
stc_regina_t.disable_on_change_activation();
'...'
stc_regina_t.calculate_now(); // calculate manually
'...'
stc_regina.deactivate(); // deactivates the trigger
'...'
stc_regina.activate(); // re-activates the trigger
```

Listing 2.11: Example: standard value change trigger with condition guard.

```
sc_event event0;
gs::av::Trigger stc_cordula_t(event0);
'...'
event0.notify(); // will trigger
```

Listing 2.12: Example: SystemC event trigger.

```
gs::av::Trigger stc_bob_t(10, SC_NS);
'...'
stc_bob_t.set_sample_interval(0, SC_NS); // removes trigger
```

Listing 2.13: Example: periodical sample interval trigger.

See files AVnewStCalc.* in the example greencontrol/examples/gav_StatCalc/ for regression tests on the trigger.

2.6.2 Calculator

Concept

The calculator functor object is responsible for performing the calculation defined by the user. The calculator is an object that inherits `calc_if` to allow to be managed by the StatCalc object. It performs the calculation when the `operator()` is called by the trigger.

The default *GreenAV calculator* object class is `Calculator`.

Interface

As an extension to the GREENAV framework new calculators can be implemented which just have to implement the virtual interface `calc_if`:

```
class calc_if {
  virtual ~calc_if() { }
  void operator()() = 0;
  gs_param_base* get_result_param() = 0;
  std::vector<gs_param_base*> get_used_params() = 0;
};
```

Listing 2.14: Calculator interface calc_if

For a detailed description see the API reference.

The function `operator()` enables the calculator object to be a functor. This function performs the calculation regardless of the formula input parameter status. The trigger has to make sure not to call the function if the calculation will fail (e.g. because of destructed formula parameters).

The function `get_result_param` returns a pointer to the result parameter. The result parameter is created and owned by the calculator. This function should not be called before the formula was specified.

The function `get_used_params` is called by the StatCalc during construction and returns all formula input parameters to be given to the trigger. These parameters are needed by the trigger for checking the destruction callbacks and – if activated – for the parameter change callbacks.

Advanced Note

Re-implement a calculator

As an extension to the GreenAV framework new calculators can be implemented. They just have to implement the `calc_if` interface.

This may be needed if other calculations shall be available, e.g. calculate within a script language instead using SystemC / C++. If only further functions consuming two parameters are needed, user defined functions can be added to the default GreenAV calculator, see below.

GreenAV default Calculator

The default *GreenAV calculator* object class is `Calculator` (and `Calculator_bit`).

Features of the Calculator:

- Variable data type of the Calculator's result and intermediate results:

  The template parameter of the Calculator class specifies the type of all operations, intermediate results and the result parameter. All formula parameters will be casted to this type for each operation.

- Formulas with various, nested operations can be built with formula input parameters (of type `gs_param<T>`):
  - Math operations $+$, $-$, $/$, $*$,
  - Logical operations $==$, $!=$, $>=$, $<=$, $<$, $>$,
  - Bitwise operations $\&$, $|$ (in the derived class `Calculator_bit`),

    ⚠ Some data types are not allowed to have bitwise operations, accordingly they are not included in the `Calculator` class. If these operations should be used, use the class `Calculator_bit` instead!

- Calc-syntax (`calc(string, gs_param<T>, gs_param<T>)`) for flexible and powerful formula creation:

  This is the main part of the calculator: the mechanism to specify formulas. Use the string to specify the operation, the input parameters may be results of another calc-call. Calls to `calc` can be nested.

- Convenient operators $+$, $-$, $/$, $*$ (less flexible and powerful but better human-readable) for parameter bases:

  The convenient operators may be used to replace the calc-syntax for the math operations.

- User defined functions for calc-syntax:

It is simple to add user defined functions including their operation strings to a Calculator of specified type. This may be done e.g. in the sc_main. A function added to `Calculator<type>` is also available in `Calculator_bit<type>`.

- Constants in formula:

  Using the calc-syntax hard-coded constants may be used instead of parameters.

- Statistics: sliding window:

  The sliding window can be activated after the formula has been specified completely. The sliding window gets a window size (*n*). The function will add the results of the last *n* calculations and divide this by the window size *n*.

- Switch to allow *sloppy calculations*:

  A special feature of the GreenAV Calculator is the ”sloppy” functionality. The user may enable sloppy behavior to pretend runtime errors, e.g. when dividing by zero. Each operation may define a default return value if sloppy is enabled. E.g. if sloppy is enabled the division (/) operation returns 0 if divided by zero.

The Calculator’s formula parameters are *limited* to these POD and SystemC *data types*:
`int`, `unsigned int`, `bool`, `double`, `float`, `unsigned long long`, `unsigned char`, `sc_int_base`, `sc_int`, `sc_uint_base`, `sc_uint`, `sc_signed`, `sc_bigint`, `sc_unsigned`, `sc_biguint`.
The data types are identified by the parameter call `getType` which returns the type as an enum gs::cnf::Param_type, see file gcnf_datatypes.h .

If parameters of type `sc_time` should be used as input parameters, a Calculator of type `double` should be used.

The Calculator does *no* initial calculation when being created! This assures that only when the specified activation events occur the calculation will be performed.

How to use

The steps before adding a calculator to the StatCalc’s constructors are:

1. Create a `Calculator` (or `Calculator_bit`) object giving a name to the constructor.
2. Specify the formula which this calculator should use. Use the calc-syntax or the convenient operators.
3. Optional: get the result parameter. This may also be called on the StatCalc object after having created it.

Calc-syntax The *calc-syntax* is the designated way of setting up a complex formula of several formula input parameters. A `calc`-call gets as function call parameters a string specifying the operation and two input parameters of type `gs_param<T>` or `gs_param_base`. Nested calls of `calc` function calls may be used because the `calc` function returns an intermediate result parameter.

The listings 2.15 and 2.16 show examples how to specify a calculation formula using the calc-syntax.

```
gs::gs_param<sc_int<10> > scint("scint", 10);
gs::gs_param<sc_biguint<70> >scbuint("scbuint", 10);

gs::av::Calculator<unsigned long long> c("my_calc");
c.calc("+", scint, scbuint);
```

Listing 2.15: Simple example using the calc-syntax:
Calculator type: unsigned long long, formula: $(scint + scbuint)$.

```
gs::gs_param<double> dbl_p("dbl_p"); gs::gs_param<int> int_p("int_p");
gs::gs_param<int> int_p2("int_p2");
gs::gs_param<unsigned int> uint_p("uint_p");

gs::av::Calculator<double> c1("my_calculator");
c1.calc("+", c1.calc("/", c1.calc("-", int_p, int_p2),
                          int_p2),
             c1.calc("*", dbl_p, uint_p));
```

Listing 2.16: Example using the calc-syntax:
Calculator type: double, formula: $(((int\_p - int\_p2)/int\_p2) + (dbl\_p * uint\_p))$.

Convenient operators Alternative to the calc function you may use more *convenient operators*: For the operations $+$, $-$, $/$, $*$ there are operators for the `gs_param_base` class. See listing 2.17 for an example, listing 2.18 shows how to mix convenient operators with calc-syntax.
You may use `c( c(i1 + i2)+ c(i2 * i2)` where i1 and i2 have to be `gs_param_bases`!

⚠ Warning: Be careful to surround each operation with a calculator call:
`my_calculator(<param_base> <operator> <param_base>)`!

⚠ Warning: Only parameter bases (`gs_param_base`) may be used,
no `gs_param<T>`!

```
gs::gs_param_base &b1 = int_p; // int_p gs_param<int>, see example above
gs::gs_param_base &b2 = int_p2; // int_p2 gs_param<int>, see example above
gs::gs_param_base &bu3 = uint_p; // uint_p gs_param<uint>, see expl. above

gs::av::Calculator<int> cc("my_convCalc");
cc(cc(b1 - cc(b1 + b2)) + cc(cc(b1 * b2) / b2));
```

Listing 2.17: Example using convenient operators:
Calculator type: int, formula: $(b1 - (b1 + b2)) + ((b1 * b2)/b2)$.

```
gs::av::Calculator<int> mc("my_mixedCalc");
mc(b1 - mc.calc("+", inp_int_par, inp_int_par2));
```

Listing 2.18: Example mixing calc-syntax and convenient operators:
Calculator type: int, formula: $(b1 - (inp\_int\_par + inp\_int\_par2))$.

Implementation Note

Convenient operators

These operators are implemented in the `gs_param_base` class. To give the user furthermore the ability to use `gs_params` simple like e.g. integers the derived classes (`gs_param<T>`) must implement the operators with normal behavior. Since only the return type differs from the calculator's operators defined in the base class the calculator's operators cannot overload the original ones. Accordingly the user must use base objects to access the calculator's operators.

Sloppy calculations: Call `enable_sloppy()` to enable the sloppy feature, call `disable_sloppy()` to disable it. Sloppy is disabled by default!

Constants Within the formula hard-coded constants may be used. Instead of giving parameters to the `calc` function, numbers may be used. See listing 2.19 for an example where 2.5 is a constant used with the calc-syntax, mixed with convenient operator syntax.

When using constants, the calc-syntax has to be used. Convenient operators do not support constants. If convenient operators should be used, instantiate `gs_params` for the constants. This produces no additional overhead because within the Calculator constants will create parameters anyway.

```
gs::av::Calculator<double> c("my_ConstCalc");
gs::gs_param_base &d = dbl_p; // dbl_p gs_param<double>, see example above
c.calc("/",   c(c(b1+b2)+c(d*b2)),   2.5); // using constant 2.5
```

Listing 2.19: Constants example: Calculator type: double, formula: $(((b1+b2)+(d*b2))/2.5)$

Bitwise operators and bool calculations When the bitwise operators `&`, `|` are needed, the `Calculator_bit`Calculator should be used. This also applies to bool calculations. Listing 2.20 shows an example using bitwise operators on integers, the example in listing 2.21 shows bool operations.

```
gs::av::Calculator_bit<int> cbit("my_BitwiseCalc");
cbit.calc("|", cbit.calc("&", int_p, int_p2), uint_p);
```

Listing 2.20: Example using bitwise operations:
Calculator type: *bit*, int, formula: $((int\_p\&int\_p2)|uint\_p)$

```
gs::gs_param<bool> bool1("bool1", true);
gs::gs_param<bool> bool2("bool2", false);

gs::av::Calculator_bit<bool> cbo("my_boolCalc");
cbo.calc("|", bool1, bool2);
```

Listing 2.21: Example bool operations: Calculator type: *bit*, bool, formula: $(bool1|bool2)$

Sliding window The *sliding window* feature can be enabled by calling `enable_sliding_window( unsigned int window_size)`. Use `window_size` = 0 for disabling the sliding window. The window size may be changed while the trigger 'runs'. You should not disable the sliding window because then the result parameter will change (you have to call `get_result_param` again).

Listing 2.22 shows an example for creating a sliding window of size 5 that is recalculated each 10 ns.

```
gs::av::Trigger tr(10, SC_NS);
gs::av::Calculator<int> calc("my_SlWiCalc");
calc.calc("+", int_p, int_p2);
calc.enable_sliding_window(5); // sliding window, size = 5
gs::av::StatCalc<int> stc("StatCalc", tr, calc);
```

Listing 2.22: Sliding window example: formula: $(inp\_p + inp\_p2)$, sliding window size: 5

Get result parameter

- The result parameter is returned when `get_result_param()` is called. The type of the result parameter is the template type of the calculator.
- After having called `get_result_param()` the formula must not be changed any longer (including enable/disable sliding window)!
- The function `get_result_param()` must not be called before a formula has been specified!
- When searching for the result parameter in the parameter list, use the parameter with the name <CalcName>_result_<highest_number>. When using a sliding window, use the parameter <CalcName>_SlidingWindow

How add user defined functions

The use can extend the calculation abilities of the GreenAV Calculator by adding functions additional to the predefined operations. The user defined functions are accessible with the calc-syntax just as the predefined ones.

- A function can (only) be added to a special template type of the `Calculator` class. The types of the function signature must match the template type.
- A function added to `Calculator<type>` is also available in `Calculator_bit<type>`.
- The calculation function must be static.
- The calculation function must match the following signature:
  `static T average(T a, T b, bool sloppy);`
- The (static) function may be placed anywhere. Only the function pointer has to be given to the add function.
- The `sloppy` parameter should be checked and if true all fatal errors that could occur during the calculation should be catched.

Listings 2.23 and 2.24 show examples how to define and add functions.

```
/// Average calculator function to be added to the Calculator<int> class
static int average(int a, int b, bool sloppy) {
  if (sloppy)
    'catch possible errors and return a default value'
  return (a + b) / 2;
}

int sc_main(int argc, char *argv[]) {
  '...'
  // Add user defined calculation function to Calculator class
  gs::av::Calculator<int>::addFunc(&average,"average");
  '...'
  sc_start();
}
```

Listing 2.23: User defined calculation function. The `average`-function will be usable by any `Calculator<int>` and `Calculator_bit<int>`.

```
/// Test class surrounding the static calculation function
class Surrounding_Class {
public:
  /// Calculator function to be added to the Calculator<double>
  static double divide_by_three(const double a, const double b, bool ↩
     sloppy) {
    return (a + b) / 3;
  }
};

int sc_main(int argc, char *argv[]) {
  '...'
  // Add user defined calculation function to Calculator class
  gs::av::Calculator<double>::addFunc(&Surrounding_Class::divide_by_three, ↩
     "div3");
  '...'
  sc_start();
}
```

Listing 2.24: User defined calculation function within class. The `average`-function will be usable by any `Calculator<double>` and `Calculator_bit<double>`.

2.6.3 StatCalc Miscellaneous

- The StatCalc's destructor deactivates the owned trigger. Keep this in mind if you use the trigger elsewhere (what you should not do).
- The Trigger and Calculator should exist at least as long as the StatCalc object! Deleting a StatCalc object does not delete the Trigger and Calculator objects (expect if the StatCalc created a trigger implicitly). If needed they have to be deleted afterwards manually.
  ⚠ Delete the trigger object *after* the StatCalc object was deleted, not before! See the GREENAV tutorial for an example.

- Input parameter may be deleted before removing or deactivating the StatCalc.
- StatCalc objects are sc_objects. To find these objects (e.g. to activate deactivated ones) a tool can search within the SystemC object hierarchy to find them.
  In feature releases the enabling shall be possible with a configurable bool parameter.

2.6.4 How to use Overview

How to use the StatCalc, the Trigger and the Calculator:

1. *Optional:* Create the Trigger (if needed other activation than formula parameter value changes)
   `gs::av::Trigger stc_t(my_event);`
2. Create the Calculator
   `gs::av::Calculator<int> stc_c("CordulaCalc");`
3. and specify the formula to be calculated
   `stc_c.calc("+", int_par, uint_par);`
4. *Optional:* Create a sliding window
   `stc_c.enable_sliding_window(5);`
5. Create a StatCalc combining the Trigger and the Calculator
   `gs::av::StatCalc<int> stc("StatCalcCordula", stc_t, stc_c);`
   or using default trigger
   `gs::av::StatCalc<int> stc("StatCalcCordula", stc_c);`.
6. *Optional:* Get the result parameter
   ```
   m_gavApi.add_to_default_output(
   gs::av::STDOUT_OUTPUT_PLUGIN, stc.get_result_param());
   ```
   or
   `gs::gs_param<int> *res = stc.get_result_param()`

See the example greencontrol/examples/gav_StatCalc for regression tests and examples on the statistics calculator, the trigger and the calculator.

2.7 Miscellaneous

2.7.1 Dynamic Processes

Important

Dynamic processes

Use `#define SC_INCLUDE_DYNAMIC_PROCESSES` before including systemc.h or use the compiler flag `-DSC_INCLUDE_DYNAMIC_PROCESSES` to enable dynamic processes needed by the GAV Plugin.

2.7.2 SystemC 2.1

Together with dynamic processes and boost tokenizer SystemC 2.1 produces compiler errors.

The parameter arrays use the boost tokenizer and this seems to have a bug in SystemC 2.1 (it seems already to be included by SystemC before GREENAV includes it in the array class).

As a work-around you have to include the boost tokenizer *before* including SystemC:

```
// Must be included BEFORE SystemC because SystemC
// has a buggy boost implementation included!!!!
#include <boost/tokenizer.hpp> // for parameter array!

#define SC_INCLUDE_DYNAMIC_PROCESSES
#include <systemc.h>
```

2.7.3 Regression Tests

See the project web page for a list of regression tests. The positive and negative regression tests are mainly done in the examples 📁 example greencontrol/examples/gav_simple/ and 📁 greencontrol/examples/gav_StatCalc/ .

In the files 🗋 simple_analysis_globals.h and 🗋 analysis_statcalc_globals.h the define `SHOW_REGRESSION_TEST_RESULTS_ON` can be undefined to switch off the regression tests output.

Appendix A

Appendix

A.1 Green Analysis and Visibility Requirements

A.1.1 General

- Capture any kind of parameters (`gs_params`) - when changed - to
  - store changes
  - make changes visible to external tools
  - calculate with other parameters (result is again a parameter)
  - calculate statistics on parameters (result is again a parameter)
- Capture parameters conditional (not only on changes)
  - boolean expression of `gs_param<bool>`s
  - event based
  - function call (manually call to statistics calculator)
  - combination of these
- The GREENAV framework does not store results itself, expect in the case of
  - statistics, e.g. sliding window
  - special output plugins that caches values
- All results, changes etc. are provided to tools, files, etc. via OutputPlugins in the GAV_Plugin.

A.1.2 Details

Requirements met by architectural design:

- captured data processed in real-time (4.1.1.2)
  - maximum data available: statistics calculators caches data they need

Functional Requirements:

- Capture API: `GAV_Api` and `StatisticsCalculator` (4.1.1)
  - Capture information if user defined condition exists: (4.1.1.1)
    - sc_event,
    - boolean expression,
    - method call,
    - combination of the three.
- Listener API `GAV_Plugin` with `OutputPlugins`
  - Tool interface: Listener API is a bridge to output types: `OutputPlugins` (4.1.2), combined with `GAV_API` (4.1.2.1) and `GCnf_Api` (4.1.2.2) (pull mechanism to browse parameters and add them to output or calculate statistics)
  - Support SCV API but do not use it (4.1.1.3), export to SCV stream(s) via OutputPlugin
- Provide information to module inside simulation (`GAV_Api`: Service Plugin API for GREENCONTROL)

  Performance requirements:

- Performance impact low per capture instance (4.3.1)
  Callbacks inside `gs_params`.
- Inactive probe negligible performance impact (4.3.2)
  Performance impact of a `gs_param` without callbacks.

A.1.3 Nice to have features

The GREENAV framework in its first version (April 2008) will provide needed functionality for several statistics. The implementation of pre-defined statistics in GreenBus may be done in the future:

- Built-in analysis parameters and statistics, e.g.
  - (4.1.1.4.1), (4.1.1.4.2), (4.1.1.4.3):
    - how many transactions generated,
    - how many certain types are generated,
    - transaction latencies,
    - utilization (in percent) of link,
    - average time in queue in the fabric,
    - what queues were full, empty, utilization of queues,
    - Transactions, ex. Addresses, data payload size
    - Analyze transactions and create statistics.
    - Register/memory accesses